Teaching an Old Robot New Tricks

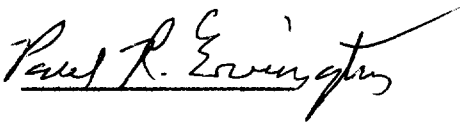
An Honors Thesis (HONRS 499)

by

Katherine Weber

Thesis Advisor

Dr. Paul Errington



Ball State University

Muncie, Indiana

July 2001

Graduation Date: July, 2001

Abstract

The process of writing an updated operation program for the RT 100 Light Industrial Robot (RTX) came in three parts. First, I worked with LabVIEW and an oscilloscope to create a method of sending hexadecimal commands to the RTX. The next step was determining what commands to send to the RTX, which was done using the original documentation and DOS-based operation program. After this was done, I put it together by writing a program in LabVIEW for moving the arm in real time and devising a method for writing easily modifiable preset motion sequences.

Table of Contents

Acknowledgements

First and foremost I would like to thank Dr. Paul Errington for all his guidance and inspiration. He has truly been a mentor to me, always ready with a helping hand when I needed one, but also knowing when to step back and be patient while I worked it out for myself. I especially want to thank him because it was having him as a teacher that brought me to this project in the first place.

Many thanks are also due to my friends and family, for their patience and support. I would lastly like to send a special thanks to Jana Stockum. Through her dedication to her own thesis project, she motivated me to overcome my own procrastination during many late nights at Cooper.

Teaching an Old Robot New Tricks

The Task

The object of my thesis project was to write an updated operation program for a stationary robotic arm. I was working with an RT 100 Light Industrial RTX Robot (henceforth referred to as the RTX), created by Universal Machine Intelligence in 1990. The original operation software for this was written in Pascal for the DOS environment. I planned to write a new program in LabVIEW to run in a Microsoft Windows environment. Through this I hoped to gain valuable experience bridging the gap between computer programming and machine control. Another benefit would be the availability of an easily modifiable program running on a modern operating system for an existing robot. This would allow the robot to be used for teaching purposes, since LabVIEW is part of the course material for the Electronics classes in the Physics curriculum at Ball State University.

The Project

The RTX operates using the COM1 serial port on a PC, which meant needing to be able to send and receive information via the serial port using LabVIEW. Luckily, there was a sample VI (VI stands for virtual instrument, and is a LabVIEW program) to demonstrate serial port communication, called Serial Communication.vi. My program is built off this foundation.

The main problem was determining how to send the RTX commands through LabVIEW, and which commands to send. I started by investigating the operation of the VI itself in both sending and receiving information. I was reluctant to test the VI on the robot itself, for fear of burning out motors, wild flailing of the arm, or some other unpredictable danger. First I

connected the serial port to an old Televideo Model 910 Terminal. Nothing happened, so I tried one of the old analog oscilloscopes that were stored in the same room as the RTX. The signal was too quick to understand by reading the screen (it showed up as a brief jump in the signal), so I connected the serial port to a digital oscilloscope.

This worked much better. I was able to "arm" the oscilloscope, so it would read in the first data it received and pause. This meant I could keep the data sent on the screen indefinitely, no matter how quickly it was sent. In this case, the VI was set to send data at 9600 bps (bits per second), which I believed would be too fast for the RTX, given its age. However, using the oscilloscope did mean that I could discern exactly how the VI was sending data.

One of the most useful things I learned at this stage was how to read binary data off an oscilloscope screen. A serial port, when unused, sits in a "high" ("1") state, at between -5 and -10 volts (in my case -9.3 volts). A "low" ("0") state is designated by a jump to between +5 and +10 volts (again in my case +9.3 volts). The voltage levels stay in that state for a certain length of time, depending on the baud rate (speed at which the data is sent). Since we were sending data as 9600 bps (bits per second), this meant that each bit was represented by a voltage pulse 104.1 microseconds (1.041×10^{-4} seconds) long. This is roughly 100 microseconds, which means that a peak around 300 microseconds wide represented three 0's in a row (000), while a trough the same width represented three 1's in a row (111).

Knowing this, I could read the binary commands straight from the oscilloscope screen. Since the least significant digit is sent earliest in time, and the time scale on an oscilloscope starts on the left, reading the screen was backwards from reading on a sheet of paper. That is, in order to read the numbers as they would be printed left-to-right (most significant to least significant), I had to read the oscilloscope right-to-left. This was a little confusing at first, but I found it quite

easy after a while, especially if I wrote them down as I measured the peaks and troughs with the oscilloscope cursors. I later learned that reading the entire display from right to left was not an entirely correct approach. This will be explained in more detail later.

Serial port commands are sent as bytes (one byte is eight bits), but I also had to keep in mind that more than just those eight bits would be sent. For each byte, a start bit ("0") appears just before the least significant digit, and the byte is followed by a stop bit ("1"). These start and stop bits appear before and after every byte, even if more than one byte was being sent in sequence. Writing down the bits as I determined them was again helpful, as I could segment the binary into bytes and note the start and stop bits.

The VI worked in ASCII characters, which was a little unfortunate since the robot worked in binary. ASCII corresponds directly to binary (a = 01100001, b = 01100010, etc.), but this was still inconvenient. Dr. Errington supplied me with a good ASCII conversion table, and I attempted to find an ASCII character for the "initialize the robot" binary command given in the RTX manual (00001000). Unfortunately, this was a backspace in ASCII. After a little more research into ASCII, I found that '\b' could also be used to represent the same binary code, but this is a special '\' code, which does not produce the same input as simply typing '\b' on an ordinary keyboard. This problem brought me back to the books, specifically the LabVIEW manual. I researched string input and found that LabVIEW supports '\' codes, as well as hexadecimal. Since I can convert from binary to hexadecimal in my head, but need a book to convert to ASCII and '\' codes, I decided that using the hexadecimal display would be the best route.

Next I attempted to send the "initialize" command mentioned earlier, which translates to 08 in hexadecimal. Nothing happened. It did not matter how many times I sent it; the robot

would not respond. Checking the oscilloscope, I found that LabVIEW was sending an extra stop bit. I scoured the VI and subVIs, setting all of them to a default value of one stop bit. I could not get rid of the stop bit, so decided to leave it for the time being. A stop bit is a "1," and the machine sits in a "1" state when idle, so an extra stop bit should not make a difference. The robot will not pay attention to what is sent after the stop bit until it is sent a "0" start bit.

A little research on LabVIEW ultimately solved this puzzle. The extra bit was a parity bit. A parity bit is used in sending commands to ensure that they arrive intact. The extra bit is added as a one or a zero to make sure the command is an even or odd number, depending on whether it is set to even or odd parity. Some of the subVIs were set to even parity, which was giving this extra bit. Setting everything to no parity solved the problem.

Since the commands in the manual were not working, I decided to go from the opposite direction. Using the original operation program that came with the robot, called Forth (or FRTX, as it will henceforth be referred), I set about finding what commands were actually being sent to the robot. I could unfortunately not capture the information sent during the initialization sequence (called init.com) because it was sending too much information too quickly to read on the oscilloscope. However, once I was inside FRTX, I could capture the individual motion commands for each of the motors on the oscilloscope and then determine the corresponding hexadecimal codes. The RTX used only one stop bit, but its arrangement was a little strange. It sent three bytes and waited between four and five milliseconds before sending the second three-byte command.

Deciphering the commands took a little bit of "backwardness" on my part. The oscilloscope shows voltage jumps and drops. When recording the binary, I would first write down as it read from left to right on the oscilloscope screen, as shown below.

...1000000001100000010010000000001...

The "...1" and "1..." were used to denote the beginning and end of the command, respectively. I would then section off the individual hexadecimal commands from the start and stop bits, to make it easier to read.

...10(00000001)10(00000100)10(00000000)1...

Here came the unexpected "backward" part of the command. I understood, as explained earlier, that I would need to read from right to left, but I had misunderstood exactly how to go about this. If one translates this sequence into hexadecimal reading left to right, one converts it incorrectly into 01 04 00. If one converts going from right to left, one will find it to be 00 20 80. This is closer to correct, but in actuality one must only translate the individual eight-digit binary numbers from right to left, and the entire sequence from left to right. So the correct conversion of this sequence into hexadecimal, as it would be entered into LabVIEW, is 80 20 00. The commands I collected from FRTX are listed in this correct form in Table 1.

Now I had commands the DOS program used and the ability to send them via LabVIEW, but still no response from the robot. Once, while sending various hexadecimal commands, the wrist began to move slowly. This was a step in the right direction, but I had no idea what the robot was doing. I was later able to repeat the motion, but the wrist reaction was too slow to know which command had caused it.

Annoyed but not yet too discouraged, I once again returned to the RTX manual. Slowly I was able to discern a sequence of commands to initialize the arm. First I entered the entire sequence into the VI, but found that it did nothing. Remembering the four millisecond delay between commands in the FRTX program, I tried entering the first two-digit hexadecimal command into the VI, running the program, entering in the second command, running the VI, and so on. It was a tedious method, but it did show that the sequence worked. Using this method, I also created a sequence of commands to start and stop soak testing. These commands are listed and explained in Table 2.

This method, as mentioned before, was very tedious, so I researched LabVIEW some more to find an easier and faster way of entering the commands. There is an object in LabVIEW called a Sequence, which is exactly that: a set of frames that execute in order as soon as its Boolean Control is set to True. This requires a button, but the default setting on a button is to toggle its value each time it is clicked. I only wanted the sequence to run once, so I had to reset the button so it would only be set as True while it was being pressed, and then pop back up to its previous value once the mouse was released. But, since my method of sending the commands was so large, I made that into a subVI, which accepted the hexadecimal code as input. I named the subVI k8serial.vi, because it handles the use of the serial port. Its terminals include Write String, Bytes to Read, Read Timeout, and Read String. Write String is the hexadecimal command to be sent to the RTX. This must be set to hexadecimal display before entering a value for the first time. If one changes the value after it has been set, it will remain in hexadecimal.

The other terminals are for receiving hexadecimal codes from the RTX. I did not explore the use of this, but I have made it available. Bytes to Read is exactly that: the number of bytes to be read by the RTX. This unfortunately must be exact. If it is set to three bytes and the RTX

only sends two bytes, the program will not register that anything has been sent until it has at least three bytes. Similarly, if the program is set to receive three bytes and receives four, part of the information is not shown. Read Timeout is the amount of time allowed to receive said number of bytes. The program will run until it has received the designated number of bytes or until the timeout, whichever comes first. Again, if too few bytes are received, the program will time out without showing any information. Finally, Read String is the string received. This is the only output terminal, and can be connected to a text box set for hexadecimal display.

Each frame of the sequence had a copy k8serial.vi with the individual command as constant input to Write String. Also in the frame was a wait command of 5 milliseconds, so enough time would elapse between each command.

The sequences worked, and I could now initialize the arm and start and stop soak testing each at the click of a button. However, even after many attempts, I was still unable to make any of the motors move individually.

On a whim, I decided to collect whatever command the robot was sent when booting up the FRTX program. This in itself was quite a bit of work, since the sequence was 52 commands long, and would therefore not all fit on a single oscilloscope screen. This meant that I had to recollect the command nearly 50 times, with the time setting shifted over by about 10 milliseconds each time so I could get the next command or two on the screen. This means that I actually could have collected the initialize sequence given in the DOS program for the RTX, but since my own initialization sequence was working fine, I decided against it.

After collecting all the commands to initialize FRTX into a single sequence, I pressed its button and, as expected, nothing happened. Then I tried clicking the Elbow Right button, then the Stop button (both of which were FRTX sequences from Table 1 and programmed into

LabVIEW for testing). It worked, and moreover it worked instantaneously. After programming all the FRTX commands into their own buttons, I found that Roll Left, Roll Right, Pitch Up, and Pitch Down still did nothing. I returned to FRTX to double check my code. It turned out that my binary was correct, but I could do nothing to get the proper response from the robot, and decided to leave that as an exercise for a future project.

The final step in this project was to program a sequence of commands into LabVIEW, so as to effectively "teach" the RTX to move via the more up-to-date program. This was done using nested sequences. The outer sequence contained the hexadecimal sequence for each command, with a Wait signal given for the time between commands. This effectively times each command so they move only for so long (and therefore only a certain distance). For instance, in my sample program, I commanded the RTX to move its shoulder left for two seconds, stop, and then move its elbow right for two seconds and then stop. Further explanation on programming motion sequences for the RTX can be found in Appendix A.

The Outcome

My final product was twofold: first, I created a "keypad" of buttons to move the RTX in real time. These work instantaneously and reliably. Second, I devised a method for writing preset motion sequences in LabVIEW. Instructions for using these programs and creating the motion sequences can be found in Appendix A. The RTX is now able to be easily programmed, and will hopefully be used in future research and classroom applications.

No research project is ever truly finished. There are other possibilities for the RTX, and I have included some of those here, for anyone who may want to continue this project. One of the most obvious tasks would be to get the roll and pitch motors to respond. Another possibility is to

create a "teach" function for the motion keypad, able to record which buttons were pressed when, and repeat the sequence with accurate timing. Finally, one might attempt to program the rest of the FRTX commands into LabVIEW, among them being variable motor speed and the "home" command, which returns the RTX to its initial position.

This project met its objectives not only in creating an easily modifiable operation program for the RTX, but it also succeeded providing a bridge between computer science and physical electronics. I not only gained experience with the RTX, the oscilloscope, and LabVIEW, but I also learned valuable research techniques and procedures, which I had been previously lacking.

Table 1

Hexadecimal motion commands sent from FRTX.

Command	First Command	Second Command
Zed Up	80 00 00	80 10 00
Zed Down	80 00 00	80 20 00
Shoulder Left	80 00 00	80 04 00
Shoulder Right	80 00 00	80 08 00
Elbow Left	80 00 00	80 01 00
Elbow Right	80 00 00	80 02 00
Roll Left	80 60 00	80 00 00
Roll Right	80 90 00	80 00 00
Pitch Up	80 50 00	80 00 00
Pitch Down	80 A0 00	80 00 00
Yaw Left	80 00 00	80 40 00
Yaw Right	80 00 00	80 80 00
Grip Open	80 00 00	80 00 01
Grip Close	80 00 00	80 00 02
Arm In	80 00 00	80 06 00
Arm Out	80 00 00	80 09 00

Table 2

RTX Procedures in Hexadecimal

Procedure	Hexadecimal Command	What Each Command Does
Initialize the Arm	20	Reset IP
This procedure resets all the motors and takes the RTX to the Home position.	00	First set of motors go active
	08	Initialize first set of motors
Note: When initializing, the arm contracts and rams the point of the gripper into the zed belt, but this does not appear to harm the robot.	29	Toggle IP (to begin sending commands to other set of motors)
	00	Other set of motors go active
	08	Initialize other set of motors

Procedure	Hexadecimal Command	What Each Command Does
Start Soak Testing	20	Reset IP
This procedure "flexes" the arm. All motors move their section of the arm to their limits on either side, to demonstrate all degrees of freedom and condition of all motors.	00	First set of motors go active.
	0A	First set of motors begin soak testing
	29	Toggle IP (to begin sending commands to other set of motors)
	00	Other set of motors go active
Note: This procedure repeats until a stop command is given.	0A	Other set of motors begin soak testing

Procedure	Hexadecimal Command	What Each Command Does
Stop Soak Testing	20	Reset IP
This procedure stops the process of soak testing.	00	First set of motors go active
	0B	First set of motors begin soak testing
Note: motors stop where they are and do not return automatically to the Home position.	29	Toggle IP (to begin sending commands to other set of motors)
	00	Other set of motors go active
	0B	Other set of motors begin soak testing

Appendix A: Operating the RTX from LabVIEW

Getting Started

National Instruments LabVIEW is required to run the operation program. Also required are the two VIs rtxsequences.vi and k8serial.vi. The first is a "keypad" of buttons. Its use is explained in "Moving the Arm" below. k8serial.vi is the subVI mentioned earlier.

To begin the program, double-click on the rtxsequences icon and set the program to Run Continuously. The RTX will not respond unless this has been done. When finished, be sure that all motors are stopped, then press the Abort Execution button in LabVIEW.

Moving the Arm

Once you have started running rtxsequences, everything is ready to go. First press the Initialize Arm button and allow it to complete its motion, at the end of which the arm will be positioned straight out. This is a slower sequence, and does not respond instantaneously. However, if the main part of the arm goes through the initialization without the wrist ever moving or vice versa, the sequence must be run again. Due to processor speeds being so much faster than that within the RTX, sometimes not all of the hexadecimal commands are read by the RTX completely.

After this has completed, press the Initialize FRTX button. The RTX will not move at this time, but this button provides the RTX will needed information to understand the motion commands.

Finally, you may press any button to move the arm as desired, in real time.* After moving each individual motor, you must press the Stop button before moving a different motor. The RTX is unable to switch motors without a Stop command in between, and is also unable to move more than one motor simultaneously using these buttons.

WARNING: Do not allow the zed (main column) to move up past the top of the column. This can burn out the motor permanently. If it reaches the top of the column and begins to make a clicking noise, flip the "kill" switch immediately and stop the program before turning the RTX back on.

Programming Motion Sequences

Since the individual motor functions (Zed Up, Elbow Left, etc.) have been created, writing a timed sequence is relatively simple. First create a Boolean Control, preferably a button rather than a switch, to run the program (these instructions are for creating a predetermined sequence of motions from the click of a single button). Right-click on the button to change the Mechanical Action of the button to "Switch Until Released." This way the program runs only once per click of the mouse.

Leave the False state of the Boolean empty. The program will reside entirely within the True state, so it only runs when the button is pressed. First put a Wait command in, allowing for enough time for the program to run. This way the button will remain in the True state for the entire duration of the program, ensuring that the entire sequence will run.*

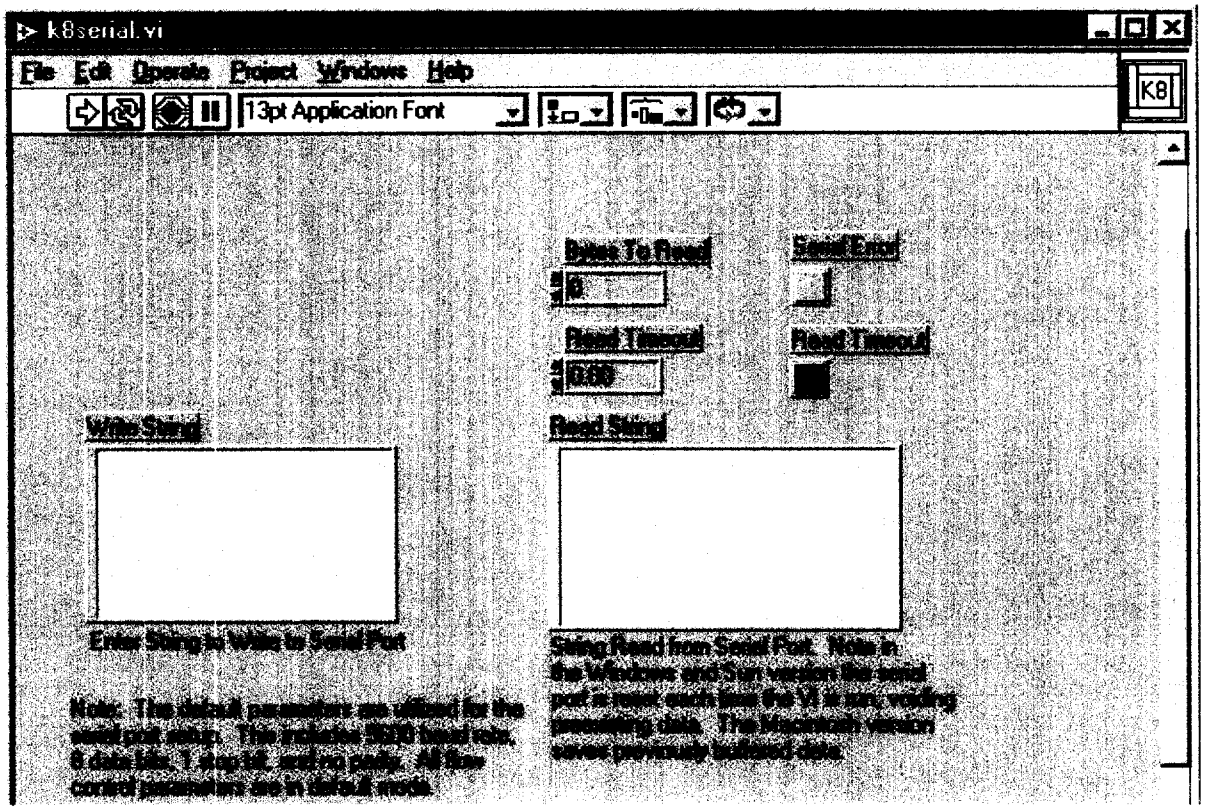
Next, put a sequence into the True state. In each frame of the sequence, copy the desired motion command sequence from rtxsequences.vi, being sure to put a stop command in every

* Because of the difference in processor speeds between the PC and the RTX, sometimes the commands do not "catch" on the first try. With the exception of the initialize sequence, if the RTX does not respond immediately, press the button a second time.

other frame. Within the frame of the motion command (but outside the motion command's individual sequence), insert a Wait command for how long you wish the motion to continue. One may also time how long a motor takes to move a certain distance, and write the program accordingly. In each stop frame, insert a Wait command of at least 1000 milliseconds, in order to give the program enough time to send the hexadecimal commands. A sample motion program is included in rtxsequences.vi.

As with regular arm motion, the Initialize Arm and Initialize FRTX sequences must be run before a program can be run. Also like the real time arm motion, the LabVIEW VI must be running continuously for the button to work.

Screen Shots



Application View of the k8serial subVI. Write String, Bytes To Read, and Read Timeout are all input terminals. Read String is an output terminal.

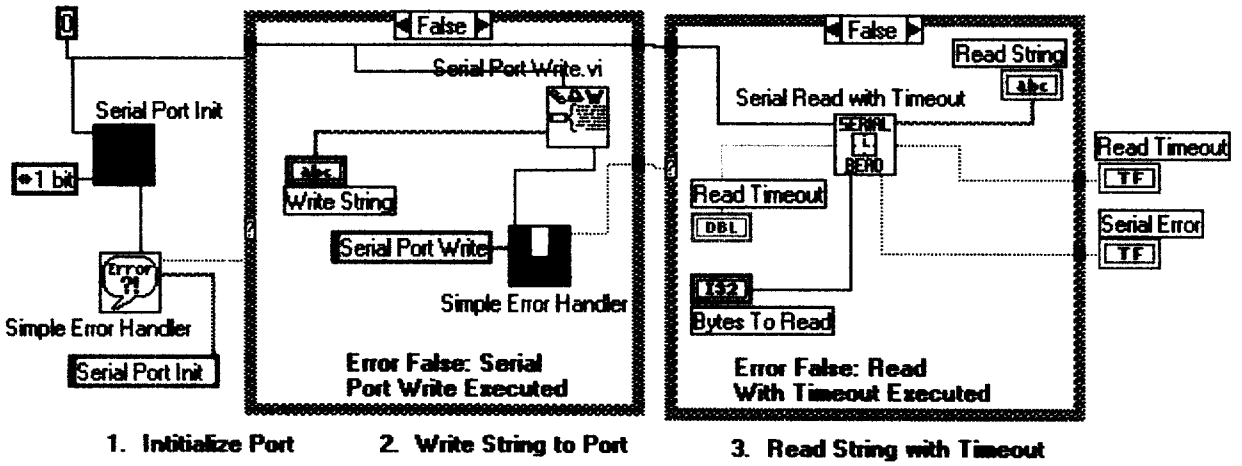
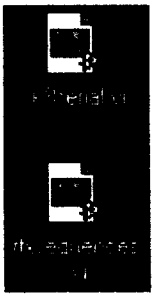
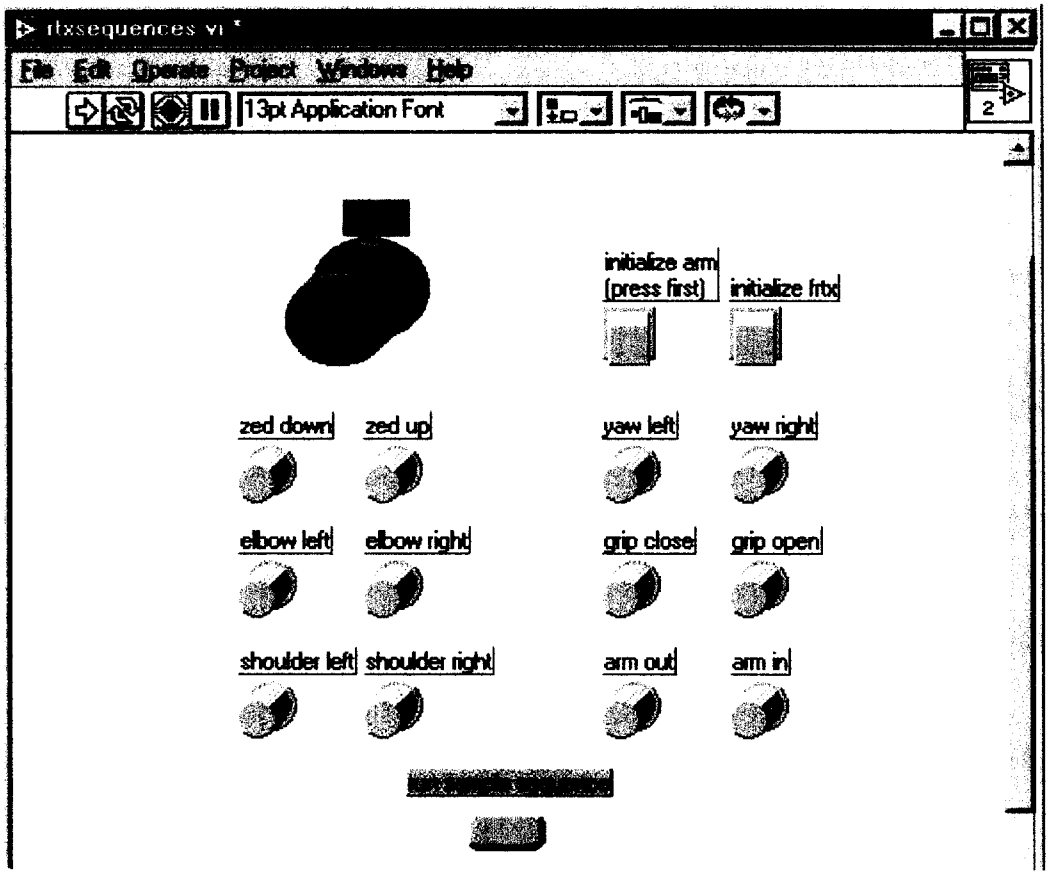


Diagram view of the k8serial subVI.



Program Icons



Application view of rtxsequences buttons.

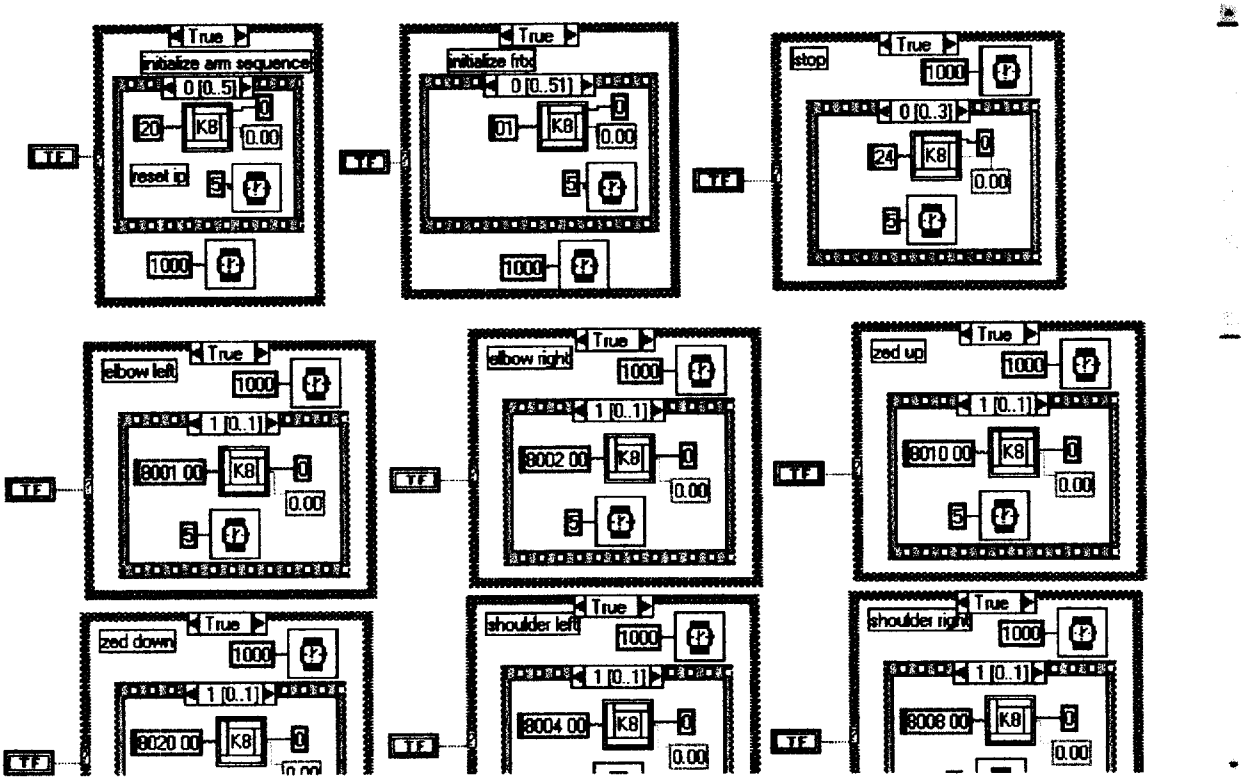
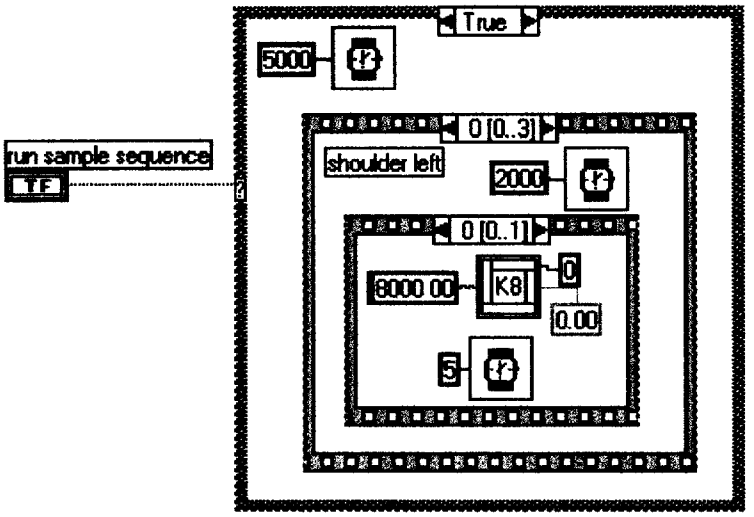


Diagram view of rtxsequences.



Sample program diagram. This program moves the shoulder to the left for two seconds (note the 2000 millisecond Wait command), stops for two seconds, moves the elbow to the right for two seconds, and then stops.